High-level Language Support for the Control of Reconfigurations in Component-based Architectures

Frederico Alvares de Oliveira Jr., Eric Rutten, Lionel Seinturier

High-level Language Support for Reconfiguration Control in Component-based Architectures

Frederico Alvares[1], Eric Rutten[1], and Lionel Seinturier[2]

[1] INRIA Grenoble, France– {frederico.alvares,eric.rutten}@inria.fr
[2] University of Lille 1 & INRIA Lille, France– lionel.seinturier@inria.fr

Abstract. Architecting in the context of variability has become a real need in todays software development. Modern software systems and their architecture must adapt dynamically to events coming from the environment (e.g., workload requested by users, changes in functionality) and the execution platform (e.g., resource availability). Component-based architectures have shown to be very suited for self-adaptation especially with their dynamical reconfiguration capabilities. However, existing solutions for reconfiguration often rely on low level, imperative, and non formal languages. This paper presents Ctrl-F, a domain-specific language whose objective is to provide high-level support for describing adaptation behaviours and policies in component-based architectures. It relies on reactive programming for formal verification and control of reconfigurations. We integrate Ctrl-F with the FraSCAti Service Component Architecture middleware platform, and apply it to the Znn.com self-adaptive case study.

1 Introduction

From tiny applications embedded in house appliances or automobiles to huge Cloud services, nowadays software-intensive systems have to fulfill a number of requirements in terms safety and Quality of Service (QoS) while facing highly dynamic environments (e.g., varying workloads and changing user requirements) and platforms (e.g., resource availability). This leads to the necessity to engineer such software systems with principles of self-adaptiveness, i.e., to equip these software systems with capabilities to cope with dynamically changes.

Component-based Architecture. Software architecture and more specifically software components have played a very important role in self-adaptiveness. Besides the usual benefits of modularity and reuse, adaptability and reconfigurability are key properties which are sought with this approach: one wants to be able to adapt the component assemblies in order to cope with new requirements and new execution conditions occurring at run-time. Component-based Architecture defines the high-level structure of software systems by describing how they are organized by the means of a composition of components [15], which are usually captured by an Architecture Description Languages (ADL). In spite of the diversity of ADLs, the architectural elements proposed in almost all of them follow the same

conceptual basis [13]. A *component* is defined as the most elementary unit of processing or data and it is usually decomposed into two parts: the implementation and the *interface*. The implementation describes the internal behaviour of the component, whereas the interfaces define how the component should interact with the environment. A component can be defined as *atomic* or *composite*, i.e., composed of other components. A *connector* mediates diverse forms of interactions of inter-component communications, and *configuration* corresponds to a directed graph of components and connectors describing the application's structure. Other elements like attributes, constraints or architectural styles also appear in ADLs [13], but for brevity we omit further details on these elements.

Initial assemblies (or configurations) are usually defined with the help of ADLs, whereas adaptive behaviours are achieved by programming fine-grained actions (e.g., to add, remove, connect elements), in either general-purpose languages within reflective component-based middleware plaforms [20], or with the support of reconfiguration domain-specific languages (DSLs)[8]. This low level of abstraction may turn the definition of transitions among configurations into a very costly task, which consequently may lead to error-prone adaptive behaviours. In fact, it may be non-trivial, especially for large and complex architectures, to obtain assurances and guarantees about the result of these reconfiguration behaviours. We claim that there is a need for a language, not only for the definition of configurations in the form of component assemblies, but also for the explicit specification of the transitions among them and the policies driving when and under which conditions reconfigurations should be triggered.

This paper presents Ctrl-F, a language that extends classic ADLs with high-level constructs to express the dynamicity of component-based architectures. In addition to the usual description of assemblies (configurations), Ctrl-F also comprises a set of constructs that are dedicated for the description of: (i) behavioural aspects, that is, the order and/or conditions under which reconfigurations take place; and (ii) policies that have to be enforced all along the execution.

Heptagon/BZR. We formally define the semantics of Ctrl-F with Heptagon/BZR [10], a Reactive Language based on Finite State Automata (FSA). It allows for the definition of generalized Moore machines, with mixed data-flow equations and automata. A distinguished characteristics is that its compilation involves formal tools for Discrete Controller Synthesis (DCS): a controller is automatically generated so as to enforce that a system behaves at runtime in concordance with the specification. The Heptagon/BZR definition of Ctrl-F programs allows to benefit from: (i) guarantees on the correctness of adaptive behaviours by either verification or control (i.e., by DCS); (ii) the compilation of adaptive behaviours towards executable code in general purpose languages (e.g., Java or C). Due to space limitation, the detailed definition is reported on elsewhere [1].

In the remainder of this paper, Sections 2 presents the self-adaptation case study Znn.com [7], used all along the paper. Section 3 presents the Ctrl-F language. Section 4 provides some details on its integration with a real component platform as well as the evaluation of its applicability through *Znn.com*. Related work is discussed in Section 5 and Section 6 concludes this paper.

2 The Znn.com Example Application

Znn.com [7] is an experimental platform for self-adaptive applications, which mimics a news website. Znn.com follows a typical client-server n-tiers architecture where a load balancer redirects requests from clients to a pool of replicated servers. The number of active servers can be regulated in order to maintain a good trade-off between response time and resource utilization. Hence, the objective of Znn.com is to provide news content to its clients/visitors within a reasonable response time, while keeping costs as low as possible and/or under control (i.e., constrained by a certain budget).

At times, the pool of servers is not large enough to provide the desired QoS. For instance, in order to face workload spikes, Znn.com could be forced to degrade the content fidelity so as to require fewer resource to provide the same level of QoS. For this, Znn.com servers are able to deliver news contents with three different content fidelity: (i) high quality images, (ii) low quality images, and (iii) only text. The objectives are: (1) Keep the performance (response time) as high as possible; (2) Keep content fidelity as high as possible or above a certain threshold; (3) Keep the number of active servers as low as possible or under a certain threshold. In order to achieve them, we may tune: (1) The number of active servers; (2) The content fidelity of each server.

As a running example for our proposal, in the next section, we extend Znn.com by enabling its replication in presence of different content providers: one specialized in soccer and another one specialized in politics. These two instances of Znn.com will be sharing the same physical infrastructure. Depending on the contract signed between the service provider and his/her clients that establishes the terms of use of the service, Znn.com Service Provider can give more or less priority to a certain client. For instance, during the World Cup the content provider specialized in soccer will always have priority over the other one. Conversely, during the elections, the politics-specialized content provider is the one that has the priority.

3 Ctrl-F Language

3.1 Overview and Common Concepts

Ctrl-F is our proposal for a domain-specific language that extends classic ADLs with high-level constructs for describing reconfigurations' behaviour and policies to be enforced all along the execution of the target system.

The abstract syntax of Ctrl-F can be divided into two parts: a static one, which is related to the common architectural concepts (components, connections, configurations, etc.); and a dynamic one, which refers to reconfiguration behaviours and policies that must be enforced regardless of the configuration.

The static part of Ctrl-F shares the same concepts of many existing ADLs (e.g., Fractal [6], Acme [13]). A *component* consists of a set of *interfaces*, a set of *event ports*, a set of *attributes* and a set of *configurations*. *Interfaces* define how a component can interact with other components. So they are used to express a

required functionality (*client interface*) that may be provided by another *component* and/or to express a provided functionally (*server interface*) that might be used by other *components*. *Event Ports* describe the events, of the given *Event Type*, a *component* is able to emit (*port out*) and/or listen to (*port in*). A *configuration* is defined as a set of *instances* of *components*, a set of *bindings* connecting *server* and *client interfaces* of those *instances* (i.e., an assembly), and/or a set of *attribute* assignments to *values*.

The dynamic part consists of a *behaviour* and a set of *policies* that can be defined for each component. A *behaviour* takes the form of orders and conditions (w.r.t. *events* and attribute *values*) under which transitions between configurations (reconfigurations) take place. The *policies* are high-level objectives/constraints, which may imply in the inhibition of some of those transitions.

The Znn.com example application of Section 2 can be modeled as a hierarchical composition of four components: *Main*, *Znn*, *LoadBalancer*, and *AppServer*. These components are instantiated according to execution conditions, the system current state (architectural composition), adaptation behaviours and policies defined within each component. Listing 1.1 shows the definition of such components with the static part of Ctrl-F.

The *Main* component (lines 1-14) encompasses two instances of *Znn*, namely *soccer* and *politics* within a single configuration (lines 7 and 8). The server interfaces of both instances (lines 9 and 10), which provides access to news services, are bound to the server interfaces of the *Main* component (lines 3 and 4) in order for them to be accessed from outside. A policy to be enforced is defined (line 13) and discussed in Section 3.3.

Component *Znn* (lines 16-33) consists of one provided interface (line 18) through which news can be requested. The component listens to events of types *oload* (overload) and *uload* (underload) (lines 20 and 21), which are emitted by other components. In addition, the component also defines two attributes: *consumption* (line 23), which is used to express the level of consumption (in terms of percentage of CPU) incurred by the component execution; and *fidelity* (line 24), which expresses the content fidelity level of the component.

Three configurations are defined for *Znn* component: *conf1*, *conf2* and *conf3*. *conf1* (lines 26-33) consists of one instance of each *LoadBalancer* and *AppServer* (lines 27 and 28); one binding to connect them (line 29), another binding to expose the server interface of the *LoadBalancer* component as a server interface of the *Znn* component (line 30), and the attribute assignments (lines 31 and 32). The attribute *fidelity* corresponds to the counterpart of instance *as1*, whereas for the *consumption* it corresponds to the sum of the consumptions of instances *as1* and *lb*. *conf2* (lines 34-39) *extends conf1* by adding one more instance of *AppServer*, binding it to the *LoadBalancer* and redefining the attribute values with respect to the just-added component instance (*as2*).

In that case, the attribute fidelity values the average of the counterparts of instances *as1* and *as2* (line 37), whereas for the consumption the same logics is applied so the consumption of the just-added instance is incorporated to the sum expression (line 38). Due to lack of space we omit the definition of configuration

conf3. Nevertheless, it follows the same idea, that is, it extends *conf2* by adding a new instance of *AppServer*, binding it and redefining the attribute values.

Listing 1.1. Architectural Description of Components *Main*, *Znn*, *Load Balancer* and *AppServer* in Ctrl-F.

```
1  component Main {
2
3    server interface sis
4    server interface sip
5
6    configuration main {
7        soccer:Znn
8        politics:Znn
9        bind sis to soccer.si
10       bind sip to politics.si
11   }
12
13   policy {...}
14 }
15
16 component Znn {
17
18   server interface si
19
20   port in oload
21   port in uload
22
23   attribute consumption
24   attribute fidelity
25
26   configuration conf1 {
27      lb:LoadBalancer
28      as1:AppServer
29      bind lb.ci1 to as1.si
30      bind lb.si to si
31      set fidelity to as1.fidelity
32      set consumption to sum(as1.
            consumption,lb.consumption)
33   }
34   configuration conf2 extends conf1 {
35      as2:AppServer
36      bind lb.ci2 to as2.si
37      set fidelity to avg(as1.fidelity
            ,as2.fidelity)
38      set consumption to sum(as1.
            consumption,as2.consumption,
            lb.consumption)
39   }
40
41   configuration conf3 extends conf2
         {...}
42
43   behaviour {...}
44   policy {...}
45 }
46
47 component LoadBalancer {
48   server interface si
49   client interface ci1,ci2,c3
50
51   port out oload
52   port out uload
53
54   attribute consumption=0.2
55 }
56
57 component AppServer {
58   server interface si
59
60   port in oload
61   port in uload
62
63   attribute fidelity
64   attribute consumption
65
66   configuration text {
67     set fidelity to 0.25
68     set consumption to 0.2
69   }
70   configuration img-ld {
71     set fidelity to 0.5
72     set consumption to 0.6
73   }
74   configuration img-hd {...}
75
76   behaviour {...}
77   policy {...}
78 }
```

Component *LoadBalancer* (lines 47-55) consists of four interfaces: one provided (line 48), through which the news are provided; and the others required (line 49), through which the load balancer delegates each request for balancing purposes. We assume that this component is able to detect overload and underload situations (in terms of number of requests per second) and in order for this information to be useful for other components we define two event *ports* that are used to emit events of type *oload* and *uload* (lines 51 and 52). Like for component *Znn*, attribute *consumption* (line 54) specifies the level of consumption of the component (e.g., 0.2 to express 20% of CPU consumption). As there is no explicit definition of configurations, *LoadBalancer* is implicitly treated as a single-configuration component.

Lastly, the atomic component *AppServer* (lines 57-78) has only one interface (line 58) and listens to events of type *oload* and *uload* (lines 60 and 61). It has also two attributes: fidelity and consumption (lines 63 and 64), just like component

Znn. Three configurations corresponding to each level of fidelity (lines 66-69, 70-73 and 74) are defined, and the attributes are valuated according to the configuration in question, i.e., the higher the fidelity the higher the consumption.

3.2 Behaviours

A particular characteristic of Ctrl-F is the capability to comprehensively describe behaviours in component-based applications. We mean by behaviour the process in which architectural elements are changed. More precisely, it refers to the order and conditions under which configurations within a component take place.

Behaviours in Ctrl-F are defined with the aid of a high-level imperative language. It consists of a set of behavioural statements (*sub-behaviours*) that can be composed together so as to provide more complex behaviours in terms of sequences of configurations. In this context, a *configuration* is considered as an atomic behaviour, i.e., a behaviour that cannot be decomposed into other *sub-behaviours*. A reconfiguration occurs when the current configuration is terminated and the next one is started. We assume that configurations do not have the capability to directly terminate or start themselves, meaning that they are explicitly requested or ended by behaviour *statements* according to the defined events and policies. Nevertheless, as components are capable to emit events, it would not be unreasonable to define components whose objective is to emit events in order to force a desired behaviour.

Statements Table 1 summarizes the behaviour statements of the Ctrl-F behavioural language. During the execution of a given behaviour B, the *when-do* statement states that when a given event of event type e_i occurs the configuration(s) that compose(s) B should be terminated and that (those) of the corresponding behaviour B_i are started.

Table 1. Summary of Behaviour Statements.

Statement	Description
B **when** e_1 **do** B_1, ... , e_n **do** B_n **end**	While executing B when e_i execute B_i
case c_1 **then** B_1, ... , c_n **then** B_n **else** B_e **end**	Execute B_i if c_i holds, otherwise execute B_e
$B_1 \mid B_2$	Execute either B_1 or B_2
$B_1 \mid\mid B_2$	Execute B_1 and B_2 in parallel
do B **every** e	Execute B and re-execute it at every occurrence of e

The *case-then* statement is quite similar to *when-do*. The difference resides mainly in the fact that a given behaviour B_i is executed if the corresponding

condition c_i holds (e.g., conditions on attribute values), which means that it does not wait for a given event to occur. In addition, if none of the conditions holds ($c_1 \wedge ... \wedge c_n = 0$), a default behaviour ($B_e$) is executed, which forces the compiler to choose at least one behaviour. The *parallel* statement states that two behaviours are executed at the same time, i.e., at a certain point, there must be two independent branches of behaviour executing in parallel. This construct is also useful in the context of atomic components like *AppServer*, where we could, for instance, define configurations composed of orthogonal attributes like fidelity and font size/color (e.g., `text || font-huge`).

The *alternative* statement allows to describe choice points among configurations or among more elaborated sequential behaviour statements. They are left free in local specifications and will be resolved in upper level assemblies, in such a way as to satisfy the stated policies, by controlling these choice points appropriately. Finally, the *do-every* statement allows for execution of a behaviour B and re-execution of it at every occurrence of an event of type e. It is noteworthy that behaviour B is preempted every time an event of type e occurs. In other words, the configuration(s) currently activated in B is (are) terminated, and the very first one(s) in B is (are) started.

Example in Znn.com We now illustrate the use of the statements we have introduced to express adaptation behaviours for components *AppServer* and *Znn* the of *Znn.com* case study. The expected behaviour for component *AppServer* is to pick one of its three configurations (*text*, *img-ld* or *img-hd*) at every occurrence of events of type *oload* or *uload*. To that end, as it can be seen in Listing 1.2, the behaviour can be decomposed in a *do-every* statement, which is, in turn, composed of an *alternative* one. It is important to mention that the decision on one or other configuration must be taken at runtime according to input variables (e.g., income events) and the stated policies, that is, there must be a control mechanism for reconfigurations that enforces those policies. We come back to this subject in Section 4.1.

Regarding component *Znn*, the expected behaviour is to start with the minimum number of *AppServer* instances (configuration *conf1*) and add one more instance, i.e., leading to configuration *conf2*, upon an event of type (*oload*). From *conf2*, one more instance must be added, upon an event of type *oload* leading to configuration *conf3*. Alternatively, upon an event of type *uload*, one instance of *AppServer* must be removed, which will lead the application back to configuration *conf1*. Similarly, from configuration *conf3*, upon a *uload* event, another instance must be removed, which leads the application to *conf2*. It is notorious that this behaviour can be easily expressed by an automaton, with three states (one per configuration) and four transitions (triggered upon the occurrence of *oload* and *uload*). However, Ctrl-F is designed to tackle the adaptation control problem in a higher level, i.e., with process-like statements over configurations.

For these reasons, we describe the behaviour with two embedded *do-every* statements, which in turn comprise each a *when-do* statement, as shown in Listing 1.3 (lines 6-14 and 8-12). We also define two auxiliary configurations: *emit-*

ter1 (line 2) and *emitter2* (line 3), which extend respectively configurations *conf2* and *conf3*, with an instance of a pre-defined component *Emitter*. This component does nothing but emit a given event (e.g., *e1* and *e2*) so as to force a loop step and thus go back to the beginning of the *when-do* statements. The main *do-every* statement (lines 6-14) performs a *when-do* statement (lines 7-13) at every occurrence of an event of type *e1*. In practice, the firing of this event allows going back to *conf1* regardless of the current configuration being executed. *conf1* is executed until the occurrence of an event of type *oload* (line 7), then the innermost *do-every* statement is executed (lines 8-12), which in turn, just like the other one, executes another *when-do* statement (lines 9-11) and repeats it at every occurrence of an event of type *e2*. Again, that structure allows the application to go back to configuration *conf2*. Configuration *conf2* is executed until an event of type either *oload* or *uload* occurs. For the former case (line 9), another *when-do* statement takes place, whereas for the latter (line 10) configuration *emitter1* is the one that takes place. Essentially, at this point, an instance of component *Emitter* is deployed along with *conf2*, since *emitter1* extends *conf2*. As a consequence, this instance fires an event of type *e1*, which forces the application to go back to *conf1*. The innermost *when-do* statement (line 9) consists in executing *conf3* until an event of type *uload* occurs, then configuration *emitter2* takes place, which makes an event of type *e2* be fired in order to force going back to *conf2*.

It is important to notice that this kind of construction allows to achieve the desired behaviour while sticking to the language design principles, that is, high-level process-like constructs and configurations. It also should be remarked that while in Listing 1.3 we present an imperative approach to forcibly increase the number of *AppServer* instances upon *uload* and *oload* events, in Listing 1.3 we leave the choice to the compiler to choose the most suitable fidelity level according to the runtime events and conditions. Although there is no straightforward guideline, an imperative approach is clearly more suitable when the solution is more sequential and delimited, whereas as the architecture gets bigger, in terms of configurations, and less sequential, then a declarative definition becomes more interesting.

Listing 1.2. AppServer's Behaviour.

```
1  component
2       AppServer {
3    ...
4    behaviour {
5      do
6       text |
7       img-ld |
8       img-hd
9      every
10      (oload
11        or uload)
12   }
13 }
```

Listing 1.3. Znn's Behaviour.

```
1  component Znn {...
2    configuration emitter1 extends conf2 { e:Emitter }
3    configuration emitter2 extends conf3 { e:Emitter }
4
5    behaviour {
6      do
7        conf1 when oload do
8          do
9           conf2 when oload do (conf3 when uload do
                 emitter2 end),
10                      uload do emitter1
11          end
12         every e2
13       end
14     every e1
15   }
16 }
```

3.3 Policies

Policies are expressed with high-level constructs for constraints on configurations, either temporal or on attribute values. In general, they define a subset of all possible global configurations, where the system should remain invariant: this will be achieved by using the choice points in order to control the reconfigurations. An intuitive example is that two component instances in parallel branches might have each several possible configurations, and some of them to be kept exclusive. This exclusion can be enforced by choosing the appropriate configurations when starting the components.

Constraints/Optimization on Attributes This kind of constraints are predicates and/or primitives of optimization objectives (i.e., maximize or minimize) on component attributes. Listing 1.4 illustrates some constraints and optimization on component attributes. The first two policies state that the overall fidelity for component instance *soccer* should be greater or equal to 0.75, whereas that of instance *politics* should be maximized. Putting it differently, instance *soccer* must never have its content fidelity degraded, which means that it will have always priority over *politics*. The third policy states that the overall consumption should not exceed 5, which could be interpreted as a constraint on the physical resource capacity, e.g., the number of available machines or processing units.

Listing 1.4. Example of Constraint and Optimization on Attributes.

```
component Main { ...
 policy { soccer.fidelity >= 0.75 }
 policy { maximize politics.fidelity }
 policy { (soccer.consumption +
           politics.consumption) <= 5 }
}
```

Listing 1.5. Example of Temporal Constraint.

```
component AppServer { ...
 policy { img-ld succeeds text }
 policy { img-ld succeeds img-hd }
}
```

Temporal Constraints Temporal constraints are high-level constructs that take the form of predicates on the order of configurations. These constructs might be very helpful when there are many possible reconfiguration paths (by either *parallel* or *alternative* composition, for instance), in which case the manual specification of such constrained behaviour may become a very difficult task.

To specify these constraints, Ctrl-F provides four constructs, as follows:

- $conf_1$ **precedes** $conf_2$: $conf_1$ must take place right before $conf_2$. It does not mean that it is the only one, but it should be among the configurations taking place right before $conf_2$.
- $conf_1$ **succeeds** $conf_2$: $conf_1$ must take place right after $conf_2$. Like in the precedes constraint, it does not mean that it is the only one to take place right after $conf_2$.
- $conf_1$ **during** $conf_2$: $conf_1$ must take place along with $conf_2$.
- $conf_1$ **between** $(conf_2, conf_3)$: once $conf_2$ is started, $conf_1$ cannot be started and $conf_3$, in turn, cannot be started before $conf_2$ terminates.

Listing 1.5 shows an example of how to apply temporal constraints, in which it is stated that configuration *img-ld* comes right after the termination of either configuration *text* or configuration *img-ld*. In this example, this policy avoids abrupt changes on the content fidelity, such as going directly from text to image high definition or the other way around. Again, it does not mean that no other configuration could take place along with *img-ld*, but the *alternative* statement in the behaviour described in Listing 1.2 leads us to conclude that only *img-ld* must take place right after either *text* or *img-hd* has been terminated.

4 Heptagon/BZR Model and Implementation

4.1 Modeling Ctrl-F in Heptagon/BZR

As architectures get larger and more complex, conceiving behaviours that respect the stated policies becomes a hard and error-prone task. This is the main reason why we model Ctrl-F behaviours and policies with Heptagon/BZR. Indeed, the FSA-based model of Heptagon/BZR allows programs to be formally exploited and verified by model checking tools [10]. The general model of Ctrl-F behaviours is as surveyed in Figure 1. Basically, each component accommodates an automaton corresponding to its adaptive behaviour, in which states correspond to configurations and transitions to reconfigurations. So, based on a vector of input events (e.g., *oload* and *uload*, in the Znn.com example) and runtime conditions (e.g., on the attribute values), transitions may be triggered while emitting signals for stopping the current configuration and starting the new one. In the case the behaviour contains choice points, that is, *alternative* statements, we model the transition conditions to each one of the choice branches as free-variables. The resulting controller from the DCS, which takes the form of a deterministic automata, is in charge of the control on those variables such that, regardless of the input events, the stated policies are enforced. It is noteworthy that although the DCS algorithms has exponential complexity as any other model checking approach, the controller is synthesized in an off-line manner and thus with no impact on the running controlled system. The same structural translation is performed hierarchically for every sub-component, i.e., in every component instantiated within another component. Due to space limitation, we have to omit the details on the translation schemes, but the full translation of Ctrl-F behaviour statements and policies to Heptagon/BZR is available in [1].

4.2 Compilation Tool-chain

As can be seen in Figure 2, the compilation process can be split into two parts: (i) the reconfiguration logics and (ii) the behaviour/policy control and verification. The reconfiguration logics is implemented by the *ctrlf2fscript* compiler, which takes as input a Ctrl-F definition and generates as output a script containing a set procedures allowing going from one configuration to another. To this end, we rely on existing differencing/match algorithms for object-oriented models [23].

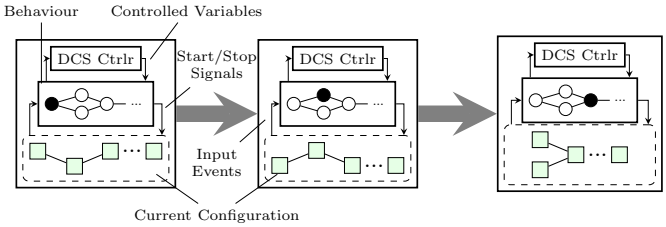

Fig. 1. Role of the Behaviour Automaton over the Transitions.

The behaviour control and verification is performed by the *ctrlf2ept* compiler, which takes as input a Ctrl-F definition and provides as output a synchronous reactive program in Heptagon/BZR. The result of the compilation of an Heptagon/BZR code is a sequential code in a general-purpose programming language (in our case Java) comprising two methods: `reset` and `step`. The former initializes the internal state of the program, whereas the latter is executed at each logical step to compute the output values based on a given vector of input values and the current state.

These methods are typically used by first executing `reset` and then by enclosing `step` in an infinite loop, in which each iteration corresponds to a reaction to an event (e.g., *oload* or *uload*), as sketched in Listing 1.6. The step method returns a set of signals corresponding to the start or stop of configurations (line 4). From these signals, we can find the appropriate script that embodies the reconfiguration actions to be executed (lines 5 and 6).

We wrap the control loop logics into three components, which are enclosed by a composite named *Manager*. Component *EventHandler* exposes a service allowing itself to be sent events (e.g., *oload* and *uload*). The method implementing this service is defined as non-blocking so the incoming events are stored in a First-In-First-Out queue. Upon the arrival of an event coming from the *Managed System* (e.g., Znn.com), component *EventHandler* invokes the step method, implemented by component *Architecture Analyzer*. The step method output is sent to component *Reconfigurator*, that encompasses a method to find the proper reconfiguration script to be executed.

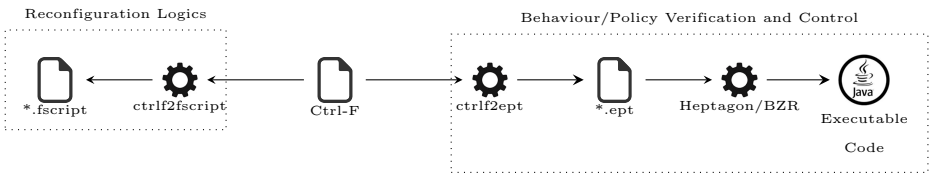

Fig. 2. Ctrl-F Compilation Chain.

Listing 1.6. Control Loop Sketch.

```
reset();
...
on event oload or uload
<...,stop_conf1,start_conf2,...>=step(oload,uload);
reconfig_script=find_script(...,stop_conf1,start_conf2,...);
execute(reconfig_script);
```

In this work, we rely on the Java-based Service Component Architecture (SCA) middleware FraSCAti [20], since it provides mechanisms for runtime reconfiguration. The FraSCAti Runtime is itself conceived relying on the SCA model, that is, it consists of a set of SCA components that can be deployed *a la carte*, according to the user's needs. For instance, in our case, the *Manager* instantiates the *frascati-fscript* component, which provides services allowing for the execution of an SCA-variant of FPath/FScript [8], a domain-specific language for introspection and dynamic reconfiguration of Fractal components. The *frascati-fscript* component relies on other components integrating the middleware, inside the FraSCAti Composite, to perform introspection and runtime reconfiguration on the managed system's components.

4.3 Adaptation Scenario

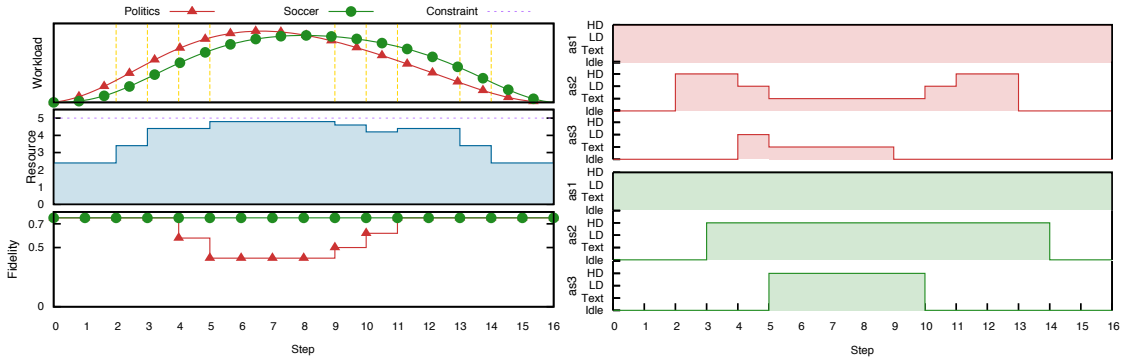

Fig. 3. Execution of the Adaptation Scenario.

We simulated the execution of the two instances of *Znn.com* application, namely *soccer* and *politics*, under the administration of the *Manager* presented in last section, to observe the control of reconfigurations taking into account a sequence of input events. The behaviours of components *AppServer* and *Znn* are stated in Listings 1.2 and 1.3, respectively, while policies are defined in Listing 1.4 and 1.5.

As it can be observed in the first chart of Figure 3, we scheduled a set of overload (*oload*) and underload (*uload*) events (vertical dashed lines), which simulate an increase followed by a decrease of the income workload for both

soccer and politics instances. The other charts correspond to the overall resource consumption, the overall fidelity, and the fidelity level (i.e., configurations *text*, *img-ld* or *img-hd*) of the three instances of component *AppServer* contained in both instances of component *Znn*.

As the workload of *politics* increases, an event of type *oload* occurs at step 2. That triggers the reconfiguration of that instance from *conf1* to *conf2*, that is, one more instance of *AppServer* is added within the *Znn*instance *politics*. We can observe also the progression in terms of resource consumption, as a consequence of this configuration. The same happens with *soccer* at step 3, and is repeated with *politics* and *soccer* again at steps 4 and 5. The difference, in this case, is that at step 4, the *politics* instance must reconfigure (to *conf3*) so as to cope with the current workload while keeping the overall consumption under control. In other words, it forces the *AppServer* instances *as2* and *as3* to degrade their fidelity level from *img-hd* to *img-ld*. It should be highlighted that although at least one of the *AppServer* instances (*as2* or *as3*) could be at that time at maximum fidelity level, the knowledge on the possible future configurations guarantees the maximum overall fidelity for instance *soccer* to the detriment of a degraded fidelity for instance *politics*, while respecting the temporal constraints expressed in Listing 1.5. Hence, at step 5, when the last *oload* event arrives, the fidelity level of *soccer* instance is preserved by gradually decreasing that of *politics*, that is, both instances *as2* and *as3* belonging to the *politics* instance are put in configuration *text*, but without jumping directly from from *img-hd*. At step 9, the first *uload* occurs as a consequence of the workload decrease. It triggers a reconfiguration in the *politics* instance as it goes from *conf3* to *conf2*, that is, it releases one instance of *AppServer* (*as3*). The same happens with *soccer* at step 10, which makes room on the resources and therefore allows *politics* to bring back the fidelity level of its *as2* to *img-ld*, and to the maximum level again at step 11. This is repeated at steps 13 and 14 for instances *politics* and *soccer* respectively, bringing their *consumptions* at the same levels as in the beginning.

The adaptation scenario is very useful to understand the dynamics behind an *Manager* that is derived from a synchronous reactive programming, which is in turn, obtained from Ctrl-F. Moreover, the scenario illustrates, in a pedagogical way, how controllers obtained by DCS are capable to control reconfigurations based not only on the current events and current/past configurations (states), but also on the possible future behaviours, that is, how controllers avoid branches that may lead to configurations violating the stated policies.

5 Related Work

In the literature, there is a large and growing body of work on runtime reconfiguration of software components. Our approach focuses on the language support for enabling self-adaptation in component-based architectures while relying on reactive systems and the underlying formal control tools for ensuring adaptation policies. This section summarizes the related work, more detailed elsewhere [1].

Classically, runtime adaption in software architectures is achieved by first relying on ADLs such as Acme [13] or Fractal [6] for an initial description of the software structure and architecture, then by specifying fine-grained reconfiguration actions with dedicated languages like Plastik [3] or FPath/FScript [8], or simply by defining Event-Condition-Actions (ECA) rules to lead the system to the desired state. A harmful consequence is that the space of reachable configuration states is only known as side effect of those reconfiguration actions, which makes it difficult to ensure correct adaptive behaviours. Moreover, a drawback of ECA rules is that, contrary to Ctrl-F, they cannot describe sequences of configurations. Even though, ECA rules can be expressed in Ctrl-F with a set of *when-do* (for the E part) and *case* (for the C and A parts) statements in parallel.

Rainbow [12] is an autonomic framework that comes with Stitch, a domain-specific language allowing for the description of self-adaptation of Acme-described applications. It features system-level actions grouped into *tactics*, which in turn, are aggregated within a tree-like strategy path. We can draw an analogy between tactics and the set of actions triggered upon a reconfiguration; as well as strategies and behaviours in the Ctrl-F language. Nonetheless, alternative and parallel, as well as event-based constructs make Ctrl-F more expressive. Furthermore, Ctrl-F's formal model enables to ensure correct adaptation behaviours.

A body of work [2][21][22][22][5][17][4] focus on how to plan a set of actions that safely lead component-based systems to a target configuration. These approaches are complementary to ours in the sense that our focus is on the choice of a new configuration and its control. Once a new configuration chosen, we rely on existing mechanisms to determine the plan of action actually leading the system from the current to the next configuration.

In [18], feature models are used to express variability in software systems. At runtime, a resolution mechanism is used for determining which features should be present so as to constitute configuration. In the same direction, Pascual et al. [19] propose an approach for optimal resolution of architectural variability specified in the Common Variability Language (CVL) [14]. A drawback of those approaches is that in the adaptation logics specified with feature models or CVL, there is no way to define stateful adaptation behaviours, i.e., sequences of reconfigurations. The resolution is performed based on the current state and/or constraints on the feature model. While in our approach, in the reactive model based on FSA, decisions are taken also based on the history of configurations which allows us to define more interesting adaptation behaviours and policies.

W.r.t. formal methods, Kouchnarenko and Weber [16] propose the use of temporal logics to integrate temporal requirements to adaptation policies. While in this approach, enforcement and reflection are performed at runtime in order to ensure correct behaviour, we rely on discrete controller synthesis.

As in our approach, in [11], authors also rely on Heptagon/BZR to model adaptive behaviours of Fractal components. However, there is no high-level description (e.g., ADL) like Ctrl-F, and reconfigurations are controlled at the level of fine-grained reconfiguration actions, which can be considered time-consuming and difficult to scale. Delaval et al. [9] propose to have modular controllers that

can be coordinated so as to work together in a coherent manner. The approach is complementary to ours in the sense that it does not provide high-level language support for describing those managers, although the authors provide interesting intuitions on a methodology to do so.

6 Conclusion

This paper presented Ctrl-F, a high-level domain-specific language that allows for the description of adaptation behaviours and policies of component-based architectures. A distinguished feature of Ctrl-F is its formalization with the synchronous reactive language Heptagon/BZR, which allows to benefit, among other things, from formal tools for verification, control, and automatic generation of executable code. In order to show the language expressiveness, we applied it to Znn.com, a self-adaptive case study, and we integrated it with FraSCAti, a Service Component Architecture middleware.

For future work, we intent to address issues of modularity and coordination of controllers, as well as their distribution. The reactive language and models we rely on have recent results that can be exploited, and can lead to deploy controllers taking into account the physical location of components.

References

1. Alvares, F., Rutten, E., Seinturier, L.: Behavioural Model-based Control for Autonomic Software Components. In: Proc. 12th Int. Conf. Autonomic Computing (ICAC'15). Grenoble, France. (extended version available as a Research Report: `https://hal.inria.fr/hal-01103548`) (Jul 2015)
2. Arshad, N., Heimbigner, D.: A Comparison of Planning Based Models for Component Reconfiguration. Research Report CU-CS-995-05, U. Colorado (Sep 2005)
3. Batista, T., Joolia, A., Coulson, G.: Managing dynamic reconfiguration in component-based systems. In: Proc. 2nd European Conference on Software Architecture. pp. 1–17. EWSA'05 (2005)
4. Becker, S., Dziwok, S., Gerking, C., Heinzemann, C., Schäfer, W., Meyer, M., Pohlmann, U.: The mechatronicuml method: Model-driven software engineering of self-adaptive mechatronic systems. In: Companion Proceedings of the 36th International Conference on Software Engineering. pp. 614–615. ICSE Companion 2014, ACM, New York, NY, USA (2014), `http://doi.acm.org/10.1145/2591062.2591142`
5. Boyer, F., Gruber, O., Pous, D.: Robust reconfigurations of component assemblies. In: Proc. 2013 Int. Conf. on Software Engineering. pp. 13–22. ICSE '13 (2013)
6. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.B.: An open component model and its support in java. In: Proc. International Symp. on Component-based Software Engineering (CBSE). Edinburgh, Scotland (May 2003)
7. Cheng, S.W., Garlan, D., Schmerl, B.: Evaluating the effectiveness of the rainbow self-adaptive system. In: Software Engineering for Adaptive and Self-Managing Systems, 2009. SEAMS '09. ICSE Workshop on. pp. 132–141 (May 2009)
8. David, P.C., Ledoux, T., Léger, M., Coupaye, T.: FPath & FScript: Language support for navigation and reliable reconfiguration of Fractal architectures. Annals of Telecommunications: Special Issue on Software Components (2008)

9. Delaval, G., Gueye, S.M.K., Rutten, E., De Palma, N.: Modular coordination of multiple autonomic managers. In: Proc. 17th Int. ACM Symp. on Component-based Software Engineering. pp. 3–12. CBSE '14 (2014)
10. Delaval, G., Marchand, H., Rutten, E.: Contracts for modular discrete controller synthesis. In: ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2010). Stockholm, Sweden (Apr 2010)
11. Delaval, G., Rutten, E.: Reactive model-based control of reconfiguration in the fractal component-based model. In: 13th International Symposium on Component Based Software Engineering (CBSE 2010). Prague, Czech Republic (Jun 2010)
12. Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. Computer 37(10), 46–54 (Oct 2004)
13. Garlan, D., Monroe, R.T., Wile, D.: Acme: Architectural description of component-based systems. In: Leavens, G.T., Sitaraman, M. (eds.) Foundations of Component-Based Systems, pp. 47–68. Cambridge University Press (2000)
14. Haugen, O., Wasowski, A., Czarnecki, K.: Cvl: Common variability language. In: Proceedings of the 17th International Software Product Line Conference. pp. 277–277. SPLC '13, ACM, New York, NY, USA (2013)
15. Jacobson, I., Griss, M., Jonsson, P.: Software reuse: architecture process and organization for business success. ACM Press books, ACM Press (1997)
16. Kouchnarenko, O., Weber, J.F.: Adapting component-based systems at runtime via policies with temporal patterns. In: 10th Int. Symp. Formal Aspects of Component Software. LNCS, vol. 8348, pp. 234–253. Nanchang, China (2014)
17. Luckey, M., Nagel, B., Gerth, C., Engels, G.: Adapt cases: Extending use cases for adaptive systems. In: Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. pp. 30–39. SEAMS '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/1988008.1988014
18. Morin, B., Barais, O., Nain, G., Jezequel, J.M.: Taming dynamically adaptive systems using models and aspects. In: Proc. 31st Int. Conf. on Software Engineering. pp. 122–132. ICSE '09, IEEE (2009)
19. Pascual, G.G., Pinto, M., Fuentes, L.: Run-time support to manage architectural variability specified with cvl. In: Proc. 7th European Conf. on Software Architecture. pp. 282–298. ECSA'13 (2013)
20. Seinturier, L., Merle, P., Rouvoy, R., Romero, D., Schiavoni, V., Stefani, J.B.: A component-based middleware platform for reconfigurable service-oriented architectures. Software: Practice and Experience 42(5), 559–583 (2012)
21. da Silva, C.E., de Lemos, R.: Dynamic plans for integration testing of self-adaptive software systems. In: Proc. 6th Int. Symp. on Software Engineering for Adaptive and Self-Managing Systems. pp. 148–157. SEAMS '11 (2011)
22. Tichy, M., Klöpper, B.: Planning self-adaption with graph transformations. In: Proc. 4th Int. Conf. on Applications of Graph Transformations with Industrial Relevance. pp. 137–152. AGTIVE'11 (2012)
23. Xing, Z., Stroulia, E.: Umldiff: An algorithm for object-oriented design differencing. In: Proc. 20th IEEE/ACM Int. Conf. on Automated Software Engineering. pp. 54–65. ASE '05 (2005)